Lab 5: Java Collection Framework, Skip List and Performance comparison

Objectives

- Getting familiar with Java collection framework
- Getting familiar with skip list
- Compare the performance of different data structure implementations
- Full mark: 40 points

Source files

- `SkipList.java`
- `Benchmark.java`
- `ExecTime.java`
- `Range.java`
- `SkipListList.java`

1 Collections

A *collection* (sometimes called a container) is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

A *collection framework* is a unified architecture for representing and manipulating collections. All collection frameworks contain the following:

- **Interfaces**: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms**: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

```
---------------------------------------------------------------------------------
| --------------------------------------------------------  ------------------- |
| |                    --------------                    |  |     -------     | |
| |                    | Collection |                   |  |     | Map |     | |
| |                    --------------                   |  |     -------     | |
| |        +-----------+-----+------+------------+      |  |  -------------  | |
| |     -------    ---------    ---------    ---------  |  |  | SortedMap |  | |
| |     | Set |    | List  |    | Queue |    | Deque |  |  |  -------------  | |
| |     -------    ---------    ---------    ---------  |  |                 | |
| |        |                                            |  |                 | |
| |  -------------                                      |  |                 | |
| |  | SortedSet |                                      |  |                 | |
| |  -------------                                      |  |                 | |
| -------------------------------------------------------  ------------------- |
--------------------------------------------------------------------------------
```

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the *Java Collections Framework*. Note that all the core collection interfaces are **generic**.

2 Skip Lists

Skip lists are designed to overcome the basic limitations of array-based and linked lists -- either search or update operations require linear time. It is also an example of a probabilistic data structure, because it makes some of its decisions at random.

As *balanced trees* (an example is the red black tree) have been used to efficiently implement `Set` and `HashMap` style data structures, skip list can be an alternative solution. Traditional balanced tree algorithms require continually rebalancing the tree, while skip list provides improved constant time overhead. Besides, search, insert and deletion are rather simple to understand and implement.

In a traditional linked list, each node contains one pointer to the next node. However, a node in a skip list contains an array of pointers. The size of this array, also called the *level* of the node, is chosen at random at the time when the node is created. For example, a level 3 node has 3 forward pointers, indexed from 0 to 2. The pointer at index 0 (called the level 0 forward pointer) always points to the immediate next node in the list, and the other pointers point to further nodes. A level`i` pointer points to the next node in the list that satisfy `level >= i`. Level of the skip list equals to the max level of all its nodes.

```
    ---                            ----                                    --
  2 | |===========================>|  |===================================>||
    |-|            ----            |--|            ----            ----    ||
  1 | |===========>|  |===========>|  |===========>|  |===========>|  |===>||
    |-|    ----    |--|    ----    |--|    ----    |--|    ----    |--|    ||
  0 | |===>|  |===>|  |===>|  |===>|  |===>|  |===>|  |===>|  |===>|  |===>||
    ---    ----    ----    ----    ----    ----    ----    ----    ----    --
   head      5       25      30      31      42      58      62      69     null
```

3 Deliverable 1 -- Skip List

Please refer to *SkipList.java*. For this lab assignment, you need to explore the Java Collections Framework to write a skip list class.

Note: it is acceptable if you use the `ArrayList` or `Vector` to store elements. However, remember a skip list is a linked list -- this means **the only element that can be directed accessed is the head**, and you have to travel through elements to find the one you need. Elements in `ArrayList` or `Vector` can be accessed by indexes.

Submit this deliverable on the eClass.

4 Deliverable 2 -- Performance comparison

As was discussed during the lectures, a way to compare different implementations is through benchmarking the different alternatives. As part of this Lab you need to benchmark your Skip List implementation. For doing it, you can use the provided `Benchmark.java` file. Since the Benchmark uses collections that implement the `List` interface, and the interface provided by SkipList.java does not implement `List`, you will use the `SkipListList.java` that works as an adapter for your SkipList class (SkipListList is an example of using the *Adapter* design pattern).

The program will generate the output of testing `ArrayList`, `LinkedList`, `Vector` and your `SkipList` implementation using `SkipListList` adaptor.

Using the output of the Benchmark program, generate a performance comparison plot (using Google sheets, Excel, etc.), one for adding elements and another for removing elements, similarly to the one presented in the Lecture.

Submit this deliverable on the eClass.

Lab 5: Java Collection Framework, Skip List

Marking sheet				
Deliverable 1 -- Skip List				
Implementation `insert` works correctly				/6
Implementation `remove` works correctly				/6
Implementation `search` works correctly				/6
Implementation `toString` works correctly				/6
Deliverable 2 -- Performance comparison				
Insert elements plot				
	The results are plotted but the performance of SkipList is worst or equal than LinkedList (1-2)	The results are plotted showing better performance of SkipList over LinkedList (3-5)		/5
Remove elements plot				
	The results are plotted but the performance of SkipList is worst or equal than LinkedList (1-2)	The results are plotted showing better performance of SkipList over LinkedList (3-5)		/5
Naming and usage of variables (in both derivables)				/2
Documentation (in both derivables)				
	No documentation (0)	One line comments but not using proper javadoc (1-2)	Clear documentation about what the function/procedure does using javadoc style (3-4)	/4
			Total:	**/40**